The Publish/Subscribe Paradigm for Scalable Group Collaboration Systems

Amit G. Mathur, Robert W. Hall, Farnam Jahanian, Atul Prakash, and Craig Rasmussen
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109.
(313) 936-2974

November 20, 1995

Contact Author: Farnam Jahanian
CSE-TR-270-95

The Publish/Subscribe Paradigm for Scalable Group Collaboration Systems

Abstract

We consider the problem of disseminating data generated in real-time to large groups of distributed users in group collaboration systems. We present an architecture based on the *publish/subscribe* paradigm to design communication services appropriate for large-scale group collaboration systems. In this model, one or more data sources or *publishers* sends data to multiple clients or *subscribers*. The distinguishing characteristic of this paradigm is the *anonymous* nature of communication. While a publisher knows about the set of subscribers, the subscribers are unaware of each other and only know about the publisher that they are receiving data from. We exploit this fact in the protocols that we use to meet the reliability and scalability requirements. We provide a formal characterization of the semantics associated with the delivery of data within this publish/subscribe framework. We then show that it is possible to use a weaker and less costly (in terms of scalability and latency) notion of synchrony in comparison to traditional work on group communication protocols. The protocols presented are based on this weaker synchrony model.

1 Introduction

With the recent explosion in the usage of the Internet, timely dissemination of data generated in real-time is gaining increasing importance. For example, ticker services that provide real-time stock quotes are becoming increasingly popular. Similarly, data generated from monitoring instruments such as radars and telescopes needs to be made available to large groups of distributed scientists and other interested users in a timely and reliable manner. Such dissemination of data gives rise to many issues, such as what are the appropriate set of semantics with respect to ordering of the information at the destinations and what kind of reliability guarantees can be made. Can these properties be achieved in a wide-area network such as the Internet that is subject to varying degrees of congestion (and hence varying latency) and varying degrees of connectivity (i.e., transient partitions)? Further, will the system be able to scale, i.e., when the data needs to be disseminated to thousands, even millions, of users, can we ensure the above properties, and yet get reasonable performance?

We have been looking at some of these issues within the context of the Upper Atmospheric Research Collaboratory (UARC) project at our institution. UARC is a multi-disciplinary effort linking research in computer science, behavioral science, and upper atmosphere and space physics [4]. We view a *collaboratory* as an advanced information environment that provides (1) human-to-human communications using shared computer tools and work spaces; (2) group access and use of a network of information, data, and knowledge sources; and (3) remote access and control of instruments for data acquisition. As part of this project we are developing a collaborative system to provide space scientists with the means to effectively view and analyze data collected by various remote instruments, present ones being located in Greenland. The goal of the UARC system is to develop groupware technologies that would not only largely eliminate the needs for costly trips to remote sites to collect data, but also provide facilities for better and more frequent collaboration between the scientists.

In the UARC system, during a scientific campaign, data generated by remote data sources such as radars, Fabry-Perot interferometers, All-Sky Imagers, IRIS riometers, and magnetometers is disseminated over wide-area networks to space scientists at their home institutions around the world (for e.g., Maryland, California, Alaska, Florida, Denmark, etc.). A " data server" gathers data from the instruments and broadcasts them to clients which run at various sites around the world. The clients then graphically display the data in various data windows.

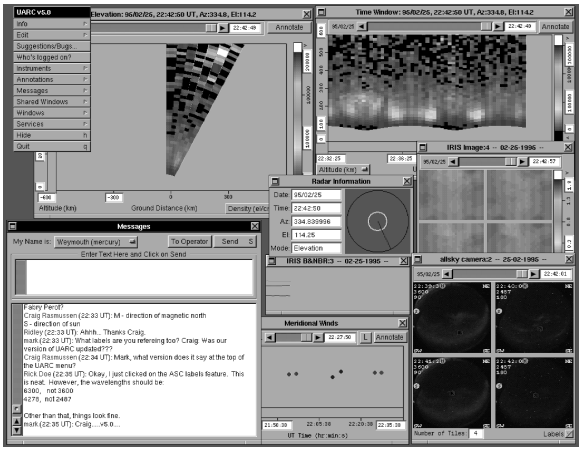


Figure 1: A snapshot of the interface presented by the multi-user UARC software to a user

Figure 1 shows the typical user interface provided by a UARC client. The three windows to the left display plots of data while the two to the right display images. Different UARC clients have access to the same data, obtained from the remote instruments, and can display different views of that data, depending on the needs of the individual scientists. A message window (bottom left in Figure 1) supports n-way textual talk.

Over the past two years, several versions of UARC (the latest being UARC 5.0) have been developed, with each prototype used by scientists in their scientific campaigns with increasing effectiveness. The usage has also led to higher demands on the system in terms of tolerance to network and system unreliability and scalability to a larger number of users. Furthermore, it has become desirable to generalize the collaborative technologies developed in UARC so that they can be used for building other collaboratories. This has led us to identify a set of "common services" that can be used to support data dissemination and tools for collaborating using this data. One can view these services as the building blocks of a collaboratory. Some of these common services include multicast communication services, cluster/server group membership and topology services, collaboration group membership services, object replication, and window sharing and annotation services.

In this paper we present an architecture and set of protocols for disseminating data generated in real-time (such as in UARC), to large numbers of users connected by wide-area networks. The data is delivered reliably and in a timely manner, and care is taken to ensure that these properties hold even when the system scales to large numbers of users. The key requirements of the system thus are *reliability*, *scalability*, and *latency*.

We present an architecture based on the *publish/subscribe* paradigm to design communication services appropriate for large-scale group collaboration systems. The basic idea of the publish/subscribe paradigm is relatively simple. Essentially one or more data sources or *publishers* sends data to multiple clients or *subscribers*. The distinguishing characteristic of this paradigm is the *anonymous* nature of communication. While a publisher knows about the set of subscribers, the subscribers are unaware of each other and only know about the publisher that they are receiving data from. We exploit this fact in the protocols that we use to meet the reliability and scalability requirements. We provide a formal characterization of the semantics associated with the delivery of data within this publish/subscribe framework. We then show that it is possible to use a weaker and less costly (in terms of scalability and latency) notion of synchrony in comparison to traditional work on group communication protocols, much of which is based on the virtual synchrony model [3]. The protocols presented are based on this

weaker synchrony model.

The rest of this paper is organized as follows: Section 2 describes the architecture and details of the publish/subscribe paradigm to support dissemination of data in group collaboration systems. Section 3 formally specifies the semantics associated with the delivery of data with respect to reliability and ordering of messages. Section 4 describes protocols to achieve these semantics. Section 5 compares our work with related work in the area, and Section 6 concludes this paper and briefly describes future work.

2 A Hierarchical System Architecture

The publish/subscribe paradigm is characterized by a publisher publishing data to multiple subscribers, where the subscribers are unaware of each other and know only about the publisher. We define a *subscription service* as the stream of messages from a publisher to a set of subscribers associated with a single application. Thus data is published for a given subscription service and subscribers can *subscribe* or *unsubscribe* to one or more such services. In the rest of the paper, we refer to a subscription service simply as a 'service'.

In order to implement this paradigm to achieve reliable dissemination of data, we propose a hierarchical architecture. In this architecture a publisher multicasts data to a set of intermediate nodes, referred to as *distributors*. The distributors then route the data to other distributors and so on until the data reaches the subscribers located at the lowest level of the hierarchy. The architecture shown in Figure 2 has a two-level hierarchy, where the publisher multicasts the data to a set of distributors. Each distributor then routes the message to its local set of subscribers. In the rest of the paper we will focus on this two-level hierarchy, although the ideas are general and can be extended to a multi-level hierarchy of distributors.

A key point of the architecture is that by introducing a hierarchy of distributors we are *minimizing system-wide knowledge and change*. A distributor has knowledge of *only* its local set of subscribers and has *no* knowledge of any of the other subscribers in the system. This allows the system to scale nicely as subscribers can join and leave without affecting the entire system. Furthermore, since distributors act as routers and multicast the messages down the hierarchy, the burden of a single source multicasting a message to a very large number of destinations and dealing with the associated ACK-implosion is alleviated to a large extent.

In summary the proposed publish/subscribe architecture is characterized by three types of processes (or active entities): publisher, distributor, and subscriber. Each process (publisher, subscriber, distributor) in this architecture is implemented on the protocol stack illustrated in Figure 3.

The *publish/subscribe group* layer provides the application programming interface for creating services, joining and leaving services, publishing and receiving messages. The *process group layer* provides process view management for subscribers and distributors, a multicast interface, message ordering and failure detection and recovery. The *transport layer* provides unreliable point-to-point transport of data packets provided by a datagram protocol such as UDP. A multicast can be a series of such point-to-point unicasts. It can also hide the exploitation of underlying multicast mechanisms such as IP multicasting [5] or hardware broadcasts available on local-area networks.

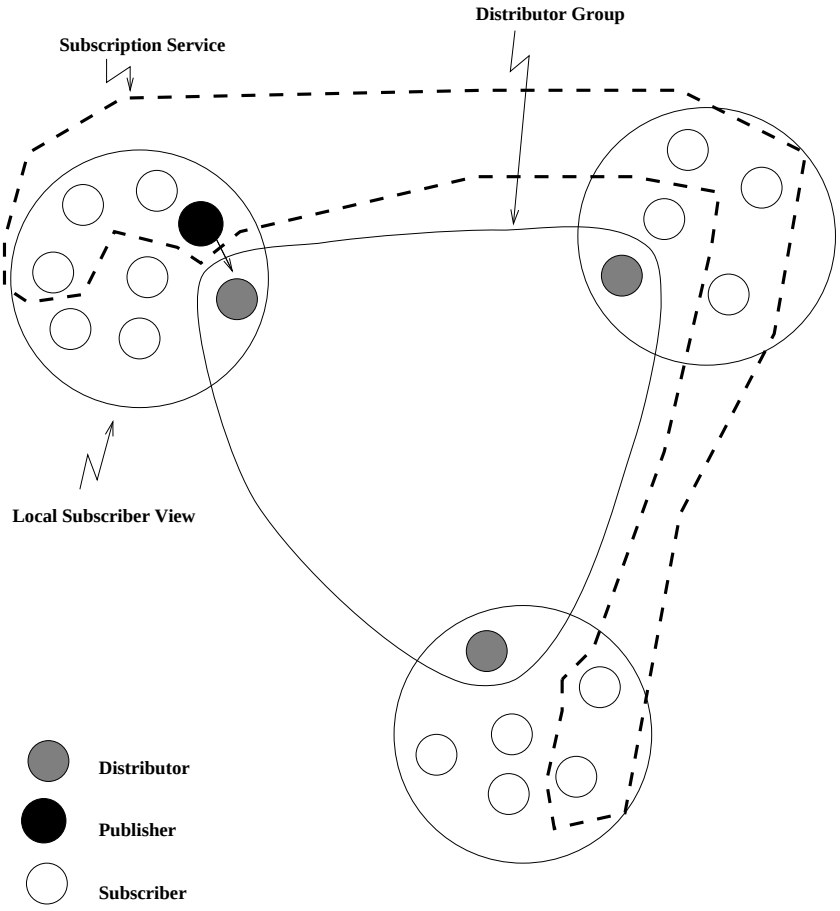


Figure 2: System Architecture

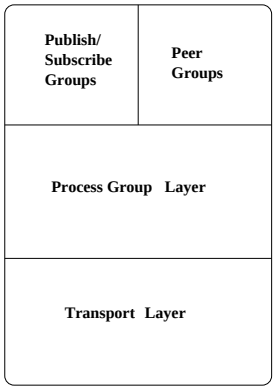


Figure 3: Protocol Stack

3 Semantics of the Publish/Subscribe Paradigm

3.1 Preliminaries

The set of processes in the system is denoted $P = \{p_1, p_2, ...\}$. As discussed previously, there are three classes of processes in the publish-subscribe model: publishers, denoted P_p, subscribers, denoted P_s, and distributors, denoted P_d. The set of processes that a process p considered to be up (formally defined below) is called the *view* of p and is denoted by $view_p$.

We make *no* assumptions regarding delay bounds about the underlying network and system, i.e., we assume that the system is asynchronous. We only consider non-Byzantine failures including communication omission and performance failures.

The execution of a process is modeled as a sequence of *primitive events*. These primitive events form the basis for defining the various ordering and atomicity constraints in the system. An event can be one of the following event types:

1. Send: The *send* event models the sending of message m by process p to process q and is denoted by $send_p(m, q)$.

2. Receive: The *receive* event models the receipt of message m by process p from process q and is denoted by $rcv_p(m, q)$.

3. View_change: The *view_change* event models the act of process p learning of an existing view and is denoted by $view\_change_p(v)$, where v denotes a view. How a process combines the received view information with its own local view is left unspecified and is determined by individual protocols.

4. Crash: The failure of process p is modeled by the local event $crash_p$. A process executes no further events after executing the crash event.

The *history* of a process p is a sequence of events executed by the process up to some *kth* event, beginning with the internal event *start*: $h_p^k =< start_p, e_p^1, e_p^2, ..., e_k^2 >$. For convenience, we will denote h_p^k by h_p when the value of k is not of particular significance. A *cut* C is an n-tuple of process histories, and is denoted $C =< h_1^{K_1}, h_2^{K_2}, ..., h_n^{K_n} >$. Events are partially ordered as defined by Lamport's happened-before relation (denoted $\rightarrow$) [8]:

1. If e_p^k, $e_p^l \in h_p$ and $k < l$, then $e_p^k \rightarrow e_p^l$.

2. If $e_p = send_p(m, q)$ and $e_q = rcv_q(m, p)$, then $e_p \rightarrow e_q$.

3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Note that it is possible that for certain events e and e', neither $e \rightarrow e'$ nor $e' \rightarrow e$. In such cases we say that e and e' are concurrent.

A cut is said to be *consistent* if for all receive events there is a corresponding send event. In the rest of the paper, when we refer to a cut, we mean a consistent cut. Informally, we use temporal logic to specify the various properties of protocols in this paper. All predicates are evaluated along consistent cuts. Given predicate ϕ and cut c the temporal logic formula $\Box\phi$ means that ϕ is true along c and all future cuts, while $\Diamond\phi$ means that ϕ holds at some future cut of the execution of the system.

Detection of process joins and failures in the system is live but not accurate since we are assuming that the system is asynchronous. This follows from the impossibility of distributed consensus in asynchronous

systems as it is impossible to distinguish with certainty between a slow and a failed process [6]. In order to formalize the properties of membership changes, we define the following predicates:
$JOINED_p(q)$: The predicate $JOINED_p(q)$ is true if $q \in view_p$, false otherwise.

$DOWN_q$: The predicate $DOWN_q$ holds on cut $c = (h_1, ..., h_q, ..., h_n)$ if $crash_q$ is the last event in h_q.

UP_q: The predicate UP_q holds on cut $c = (h_1, ..., h_q, ..., h_n)$ if $crash_q$ is not an event in h_q, and $JOINED_q(q)$. Note that when a process crashes and comes back up it has a process identifier different from its original process identifier.

The membership changes satisfy the following properties:

1. (Failures/Leaves) If q crashes, then eventually either p removes q from its view or p crashes: $DOWN_q \Rightarrow \Diamond(\neg JOINED_p(q) \vee DOWN_p)$.
2. (Joins) If q is up, then eventually either p adds q to its view or p crashes: $UP_q \Rightarrow \Diamond(JOINED_p(q) \vee DOWN_p)$.
3. (Reciprocity) If p suspects q is inaccessible, then if q does not crash, it eventually suspects p is inaccessible: $\neg JOINED_p(q) \Rightarrow \Diamond(\neg JOINED_q(p) \vee DOWN_q \vee JOINED_p(q))$.

To simplify presentation of the semantics below, we assume, without loss of generality, that there is a single publisher in the system. The case where there are multiple publishers/services is a straightforward extension wherein each distributor d maintains a separate view for each service.

To model send and receive events, we define the following predicates:
$SEND_p(m, q)$: The predicate $SEND_p(m, q)$ is true on cut $c = (h_1, ..., h_p, ..., h_n)$, if $send_p(m, q)$ is the last event in history h_p.

$RCV_p(m, q)$: The predicate $RCV_p(m, q)$ is true on cut $c = (h_1, ..., h_p, ..., h_n)$, if $rcv_p(m, q)$ is the last event in history h_p.

We next define the following partial ordering on views (denoted $<$) [7]:

1. Let $view$ and $view'$ be two views. If $view = view_p^x$ and $view' = view_p^{x+1}$, then $view \not< view'$.
2. (Irreflexivity) $view \not< view$.
3. (Asymmetry) If $view < view'$ then $view' \not< view$.
4. (Transitivity) If $view < view'$ and $view' < view''$, then $view < view''$.

An execution of a membership protocol generates a set of views. The weak membership protocols, in [7] and [15], generate a set of views such that the views are partially ordered by the relation $<$ described above. On the other hand, the set of views generated by the strong group membership protocols, such as those described in [14, 13, 7, 15], satisfy the partial ordering $<$, but in addition satisfy the following two properties:

1. Mutual Exclusion and Concurrent Views: If $view \not< view'$, then $\forall p \in view \Rightarrow p \notin view'$.
2. Total Order on View Changes: There exists a predicate π defined on the set of views which converts the partial ordering into a total ordering on a subset of the views. An example of the predicate π is a majority, where views with majority of processes are totally ordered.

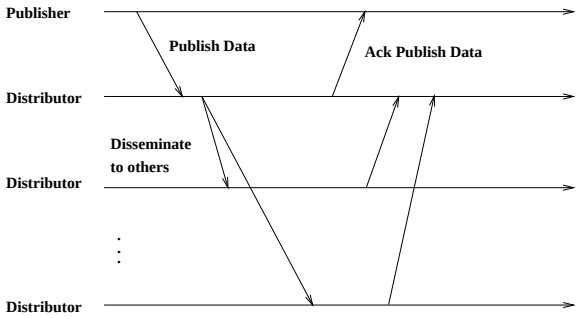


Figure 4: Publish Data for a Service

3.2 The Semantics

We are now ready to give the formal semantics of the primitives that constitute the publish/subscribe paradigm.

In the following we denote a subscriber process by s ($s \in P_s$), a publisher process by pub ($pub \in P_p$), a distributor process by d ($d \in P_d$). A distributor d's view of the set of distributor processes is denoted $view\_dist_d$, and its view of its subscriber processes is denoted $view\_sub_d$. A subscriber s's view of the distributors is denoted $view\_dist_s$. Equivalently, a publisher p's view of the distributors is denoted $view\_dist_p$. These views that publishers/subscribers maintain is not explicitly maintained by a membership protocol, but instead represents a *core group of distributors* that are likely to be up [7], and can be obtained from a well-known located . This set of processes consists of a primary distributor and some other likely secondary distributors.

Since a publisher/subscriber has no knowledge of the other publishers/subscribers in the system, it maintains no view of the other publishers/subscribers in the system.

- **Publish to a service:**
  A publisher pub sends a publish message m to distributor d. Upon receiving m, d sends m to each $d_i \in view\_dist_d$. Upon receiving m, each d sends m to each $s_i \in view\_sub_{d_i}$. This is illustrated in Figure 4.

  1. If publisher pub sends message pub_msg to a distributor d in pub's distributor view, we guarantee that eventually either d receives pub_msg or pub is down or d is down.

  $$(SEND_{pub}(pub\_msg, d) \wedge d \in view\_dist_{pub}) \Rightarrow$$
  $$\Diamond(RCV_d(pub\_msg, pub) \vee DOWN_{pub} \vee DOWN_d)$$

  2. If d receives pub_msg from pub, then eventually each d' in d's view will receive pub_msg or will be down. '*' below indicates d' doesn't care where it receives pub_msg from.

  $$\forall d' \in view\_dist_d, RCV_d(pub\_msg, pub) \Rightarrow$$
  $$\Diamond(RCV_{d'}(pub\_msg, *) \vee DOWN_{d'})$$

  Note that this does not guarantee atomic delivery from the publisher to the distributors. All it guarantees is that once it gets from the publisher to any one correct distributor, then all other correct distributors will get it. Note also that if pub and d fail right after p sends m to d, then this is indistinguishable from p failing before it sent m.

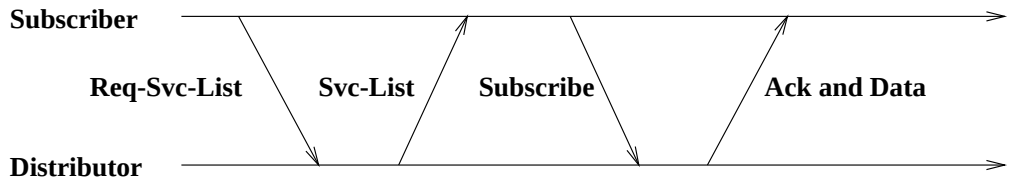
3. A publisher generates a sequence of messages, each having a sequence number one more than the previous message. Messages are delivered in this order (sender-based FIFO) at both distributors and subscribers.
4. There may be a *gap* in the sequence of messages from a distributor as seen by a subscriber. Two semantics are possible for delivery of messages to subscribers, with different scalability implications as shown later. One possible semantic tolerates gaps in a message stream from a distributor as seen by a subscriber, the other does not.
   (a) Gap semantic: If d receives pub_msg then eventually s receives pub_msg or s is down or d is down, i.e.,
   $$(RCV_d(pub\_msg, *) \wedge s \in view\_sub_d) \Rightarrow$$
   $$\Diamond(RCV_s(pub\_msg, d) \vee DOWN_s \vee DOWN_d)$$
   A gap can occur in the message sequence received by s because the above semantic only ensures that s receives the messages if d does not crash.
   (b) No gap semantic: If d receives pub_msg then eventually s receives pub_msg or s is down, i.e.,
   $$(RCV_d(m, *) \wedge s \in view\_sub_d) \Rightarrow$$
   $$\Diamond(RCV_s(pub\_msg, d) \vee DOWN_s)$$
   A gap does not occur in the message sequence received by s because the above semantic ensures that s receives the message even if d does crash.

- **Subscribe to a service:**

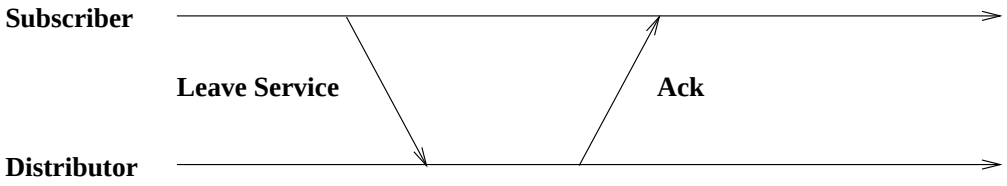
  A subscriber s sends a subscribe message, $send_s(sub\_msg, d)$, to its primary distributor, d. We support two different semantics, depending on requirements, and with different scalability assumptions.

  1. If d receives subscription request sub_msg from s, and for all future published messages pub_msg that d receives, eventually s receives pub_msg or s is down or d is down or s has unsubscribed:
  $$(RCV_d(sub\_msg, s)) \Rightarrow$$
  $$\Box((\forall pub\_msg, RCV_d(pub\_msg, *) \Rightarrow$$
  $$\Diamond(RCV_s(pub\_msg, *) \vee DOWN_d \vee DOWN_s \vee RCV_d(unsub\_msg, s)))$$
  2. If d receives subscription request sub_msg from s, then for all future published messages pub_msg that d receives, eventually s receives pub_msg or s is down or s has unsubscribed:
  $$(RCV_d(sub\_msg, s)) \Rightarrow$$
  $$\Box((\forall pub\_msg, RCV_d(pub\_msg, *) \Rightarrow$$
  $$\Diamond(RCV_s(pub\_msg, *) \vee DOWN_s \vee RCV_d(unsub\_msg, s)))$$

  Upon receipt of a subscribe message, $rcv_d(sub\_msg, s)$, the distributor d adds this subscriber to its subscriber view, $view\_sub_d$. Since s is now in $view\_sub_d$, any data messages delivered to d will now be sent to s. d sends an acknowledgment message to s, as s needs this to publish or receive messages. This is illustrated in Figure 5a.



(a) Subscribing to a Service



(b) Unsubscribing from a Service

Figure 5: Subscribing and Unsubscribing a Service

- **Unsubscribe from a service:**

  A subscriber s sends an unsubscribe message, $send_s(unsub\_msg, d)$, to its primary distributor, d. Formally we specify this as:

  $$(SEND_s(unsub\_msg, d)) \Rightarrow$$

  $$\Diamond(s \notin view\_sub_d \vee DOWN_d)$$

  Upon receipt of an unsubscribe message, the distributor d removes this subscriber from its view, $view\_sub_d$. Since s is no longer in $view\_sub_d$, any future data messages delivered to d are not delivered to s. Lastly, d sends an acknowledgment to s. This is illustrated in Figure 5b.

- **Create a service:**

  To publish to a new service, a publisher must create the service first. The publisher p that does this is referred to as the service *creator*. It sends a create message, $send_p(create\_service\_S, d)$ to its primary distributor d, where S is the new service. d establishes a view for the service that contains only the publisher initially. d sends a *publicize* message to each $d' \in view\_dist_d$ to notify them of this new service. Each receiving d' creates a local subscriber view for the service, initially empty.

4 Protocols for Publish/Subscribe

The protocols that implement the publish/subscribe primitives described above address two related issues of *view management* and *multicasting*. View management involves maintaining state about processes in the system and changes to that state. Multicasting involves sending messages from one process to processes in its view.

4.1 View Management

We describe view management at a subscriber process and at a distributor process. View management at a publisher process is handled in a similar way to the view management at a subscriber.

4.1.1 View Management at a Subscriber

Each subscriber connects to a distributor referred to as the primary distributor, and receives its data from this distributor. As described in section 3.2, the subscriber maintains a view of the set of distributors in the system ($view\_dist_s$). The other distributors, referred to as secondary distributors, are available as backups that the subscriber can connect to if the primary distributor were to fail. The subscriber obtains this list from a well-known location. This list may not represent the most recent view of the distributors.

Recall that in the publish/subscribe model, for scalability reasons, a distributor knows only about its local set of subscribers and has no knowledge of the other subscribers in the system. Furthermore, each subscriber is responsible for detecting the failure of the distributor to which it is connected and for connecting to another distributor. Hence, in the presence of distributor failures,distributors have no way of ensuring that each subscriber in the system has a consistent view of the current set of functioning distributors. For example, consider the case where a distributor with one or more subscribers connected to it crashes. Knowledge of the set of subscribers connected to this distributor is lost from the system with the crash of the distributor. So when the remaining functioning distributors reconfigure and a new view gets established, the set of subscribers that were connected to the failed distributor have no way of being informed about this view other than by doing a look up at the well-known location where this information may be posted. In other words, subscribers do not run an agreement protocol to maintain a consistent view of the distributors.

We can now describe how a subscriber, s, reconfigures after it detects the failure of its primary distributor, d:

1. s removes d from its view of distributors $view\_dist_s$, i.e., $view\_dist_s = view\_dist_s - \{d\}$.

2. Optionally, s attempts to connect to one of the secondary distributors, $d', d^{'} \in view\_dist_s$. Otherwise, s obtains a new $view\_dist$ as described in 4 below.

3. If s is successful in connecting to a distributor d' then d' adds s to its set of local subscribers, i.e., $view\_sub_{d'} = view\_sub_{d'} \cup \{s\}$. s then begins to receive messages from d'.

4. If s is unable to connect to any of the distributors, then either all distributors are unavailable or s's view of distributors is stale and it needs to be updated by obtaining a more recent version from the well-known location.

Note that it is possible s may have incorrectly suspected its original distributor of crashing. In such a situation, s will be in the view of both the original distributor and the new distributor that it connected to, and will receive messages from both. s will then close its connection to one of the two distributors by unsubscribing to it. Notice that subscribers can make decisions regarding failure of their primary distributor without reaching any sort of agreement with other processes in the system. This is a key design decision for achieving scalability as the number of subscribers increases in a system.

4.1.2 View Management at a Distributor

As defined formally in section 3.2, each distributor process d maintains two views: the view consisting of the set of local subscribers (denoted by $view\_sub_d$) and the view consisting of the other distributors in the system (denoted by $view\_dist_d$).

View changes to $view\_sub_d$, the set of local subscribers, are handled strictly locally without involving the other distributors. As subscribers join and leave, the distributor respectively adds and removes them from $view\_sub_d$. When the distributor detects the failure of a subscriber s, s is simply removed from $view\_sub_d$, as defined formally in section 3.2.

View changes to $view\_dist_d$, the set of distributors, are done cooperatively among the set of distributors, i.e., agreement is reached amongst the distributors before there can be a change in the membership of $view\_dist_d$ at any distributor. This ensures that membership changes to $view\_dist_d$ are totally ordered at each distributor. The protocol to achieve this is exactly the strong group membership protocol presented in the literature [14, 13, 7, 15]. Briefly, a common protocol for strong group membership operates by designating a distributor process as the coordinator of the set of distributor processes. The coordinator d_c initiates changes in $view\_dist_d$ by executing a 2-phase protocol, as follows:

- In phase one, d_c sends a message to all other distributors proposing a new membership for $view\_dist_d$.
- d_c gathers acknowledgments from the distributors.
- In phase two, d_c sends a view-change message to each of the processes that acknowledged and gives them the new agreed-upon membership of $view\_dist_d$.

This protocol is also able to handle the case when the coordinator crashes during execution of the 2-phase view change protocol. Essentially a new coordinator is elected that takes over the responsibility of the failed coordinator.

4.2 Multicasting

There are two kinds of multicasts in the system. The first kind, referred to as a *distributor multicast* involves the multicast of messages received by a distributor from a publisher to all other distributors in the system. The second kind of multicast, referred to as a *subscriber multicast*, involves the multicast of messages received by each distributor to its local view of subscribers.

4.2.1 Distributor Multicast

The semantics specified for the distributor multicast in section 3.2 requires that if a message is received by any distributor, then as long as that distributor stays functional, all correct distributors will receive that message. To ensure this, distributors need to buffer messages until they are *stable*. [1] A message is said to be stable amongst a group of processes when each process knows that the message has been received by every other process in the group. A process can discard a message once it becomes stable. The group of distributor processes exchange periodic messages amongst each other indicating the highest message sequence number received. When there is a distributor failure and a new view is established as described above, messages do *not* need to be flushed from the previous view before any new messages can be delivered in the new view. All our semantics requires is that messages get delivered *eventually*.

[1] For a formal definition of stability see [2] and the references therein.

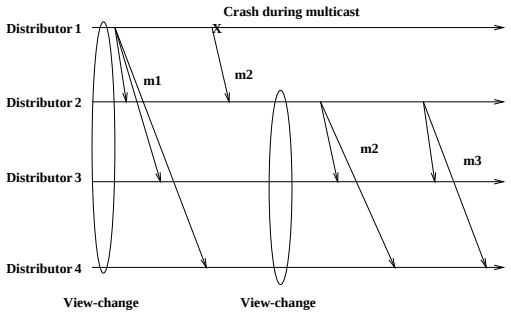


Figure 6: Scenario to illustrate the weaker notion of synchrony used in the publish/subscribe paradigm

They could get delivered at two distributors that progressed from a given group to the next group in different views.

This differs from typical flush protocols used by systems such as Isis and Transis which ensure the virtual synchrony property [3, 11]. Here we exploit the semantics of the publish/subscribe paradigm to give a *weaker* notion of synchrony as compared to virtual synchrony ([3]) and its variant, extended virtual synchrony ([11]).

To illustrate this weaker synchrony, consider the scenario shown in Figure 6. The initial group of processes includes distributors 1 through 4.

1. Distributor 1 multicasts message $m1$ to the entire group.
2. During the multicast of $m2$, distributor 1 crashes after sending the message to distributor 2, but before it is able to send it to distributors 3 and 4.
3. The failure of 1 is detected and a new group is established consisting of 2, 3, and 4.
4. Distributor 2 forwards $m2$ to 3 and 4, but this happens once the new view has been established.

Thus message $m2$ is delivered in different views by distributor 2 and distributors 3 and 4. In the virtual synchrony model, the new view would not be established until distributor 3 and 4 also delivered $m2$ in the old view.

4.2.2 Subscriber Multicast

In section 3.2 we defined two possible semantics for the multicast of messages from the distributor to its set of subscribers. The first semantic tolerated gaps in the message stream from the distributor to a subscriber, while the other did not. Let us consider each semantic and see its impact on the multicast protocols.

The first semantic ensures that if a distributor receives a message m, then as long as both the distributor and the subscriber stay up, the subscriber will receive the message. This is easy to ensure. The distributor simply multicasts the messages it receives to its set of subscribers. If the distributor fails during any of these multicasts, the subscribers may not receive the message, leading to a possible gap in the sequence of messages received. After the subscriber connects to a new distributor, it will start receiving messages from that distributor. It is possible that the new distributor may still have the messages that a subscriber missed, but this cannot be guaranteed with certainty. [2].

[2] Note that if were to assume infinite buffering, then the problem of gaps is easily solved.

The second semantic guarantees that there will be no gaps in the sequence of messages received at a subscriber even if its primary distributor fails. In this case each distributor may need to buffer a message until it knows that *every* subscriber in the system has received it. This implies that a distributor needs to maintain global membership information about subscribers at other distributors in the system. This imposes a severe limitation on the scalability of the system.

Note that subscribers do not need to buffer any messages as they will never be called upon to forward that message to some other process in the system for stability reasons.

5 Related Work

The publish/subscribe paradigm as a basis for structuring dissemination-style communication has been considered in earlier work. Oki et al [12] use the paradigm to develop the notion of an Information Bus which acts as an intermediary between the publisher and subscriber. In our work, the hierarchy of distributors connects the publisher to the subscribers. Further, Oki et al do not address aspects of fault-tolerance related to maintaining message delivery and membership atomicity. Rajkumar et alsha:95 have extended the publish/subscribe paradigm to support distributed real-time applications although they do not address fault-tolerance aspects.

Systems such as Isis [3], Consul [9, 10], Transis [1], and Horus [16] are based on the virtual synchrony model and are geared more towards a general class of fault-tolerant distributed applications. We have focussed on a smaller class of distributed applications, viz., dissemination of data in group collaboration systems, and by exploiting semantics of this application class have developed a weaker synchrony model that allows for protocols that can *scale* better than protocols within the virtual synchrony framework.

6 Conclusions and Future Work

The publish/subscribe communication model described in this paper allows for dissemination of data in group collaboration applications to a large set of distributed subscribers. Scalability is addressed through identifying distinct roles of publishers, distributors, and subscribers and the lack of globally shared state that results from view management properties associated to these roles. Reliability is addressed by providing guarantees on message delivery that arise from the weak form of synchrony we provide.

The UARC software (UARC 5.0) has been implemented and is in use by space scientists at many sites in Europe and North America. The next-generation (UARC 6.0) is in the prototype stage. The prototype system implements the proposed publish/subscribe architecture and protocols. The implemented transport layer provides FIFO, reliable datagram multicast over UDP sockets in a UNIX environment, specifically SunOS 4.2.X, HP-UX, and NeXTStep Mach. The process group layer implementation provides distributor view management based on a strong group membership protocol for a group of distributor processes, as well as view management for publishers/subscribers and the capability of providing multiple subscription services. A C++ publish/subscribe application interface library for use by publisher and subscriber applications has been implemented and is being used by UARC applications.

Acknowledgments

This work is supported in part by the National Science Foundation under cooperative agreement IRI-9216848.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. Technical Report TR CS91-13, Computer Science Dept., Hebrew University, April 1992.

[2] O. Babaoğlu and K. Marzullo. *Distributed Systems*, chapter Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Addison-Wesley, 1993.

[3] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, Feb. 1987.

[4] C. R. Clauer, D. E. Atkins, T. E. Weymouth, G. M. Olson, R. Niciejewski, T. Finholt, A. Prakash, C. E. Rasmussen, T. J. Rosenberg, J. D. Kelly, Y. Zambre, P. Stauning, E. Friis-Christensen, and S. B. Mende. A Prototype Upper Atmospheric Research Collaboratory (UARC). *EOS, Trans. Amer. Geophys. Union*, 74, 1993.

[5] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Trans. on Computer Systems*, 8(2):85–110, May 1990.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1995.

[7] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor Group Membership Protocols: Specification, Design, and Implementation. In *Proc. of Symposium on Reliable Distributed Systems*, Oct. 1993.

[8] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558–565, July 1978.

[9] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol Based on Partial Order. In *Proc. of the 2nd. Intl. Conf. on Dependable Computing for Critical Applications*, pages 309–332, Tucson, AZ, Feb. 1991.

[10] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineeering Journal*, 1(2):87–103, Dec. 1993.

[11] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.

[12] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus-An Architecture for Extensible Distributed Systems. In *Proc. of the ACM Symposium on Operating Systems Principles*, pages 58–68, North Carolina, December 1993.

[13] A. M. Ricciardi. The Group Membership Problem in Asynchronous Systems. Technical Report TR92-1313, Computer Science Dept., Cornell University, Nov. 1992.

[14] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of 10th. Annual ACM Symp. on Principles of Distributed Computing*, Aug. 1991.

[15] A. Schíper and A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. In *Proceedings on the 13th International Symposium on Fault-Tolerant Computing*, pages 534–568, Toulouse,France, June 1993.

[16] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR94-1442, Computer Science Dept., Cornell University, Aug. 1994.